



資訊管理學系 陳士杰老師

作業系統

Operating Systems

死結

Dead Lock



國立聯合大學
NATIONAL UNITED UNIVERSITY



■ 本章重點

- ⊕ **Def. of Dead Lock**
- ⊕ **與Starvation的比較**
- ⊕ **死結的處理方式**
 - ⊕ **Deadlock Prevention**
 - ⊕ **Deadlock Avoidance**
 - ⊕ **Deadlock Detection and Recovery**
- ⊕ **相關定理 for Dead Lock Free**



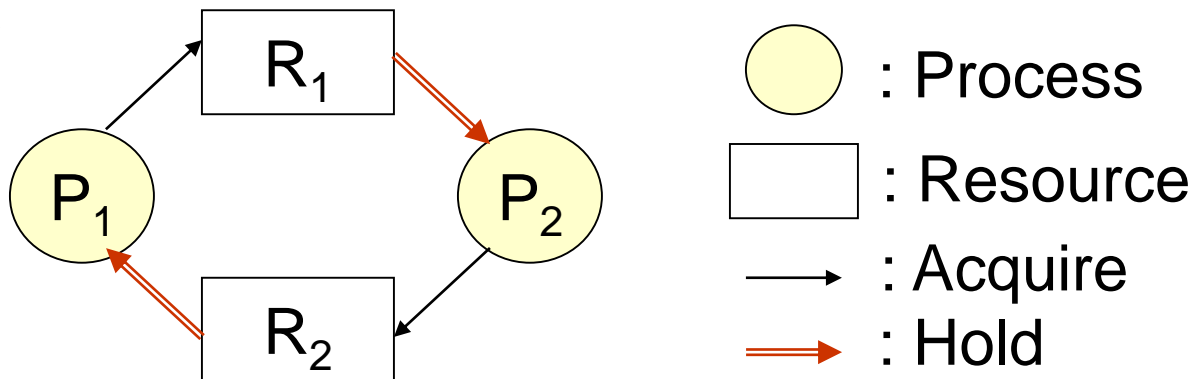


Dead Lock

- Def: 系統中存在一組Processes陷入“**互相等待對方所擁有之資源**”的情況 (即: **Circular Waiting**)，造成所有Processes皆無法往下執行，使得**CPU** 利用度及產能大幅降低。

- Example

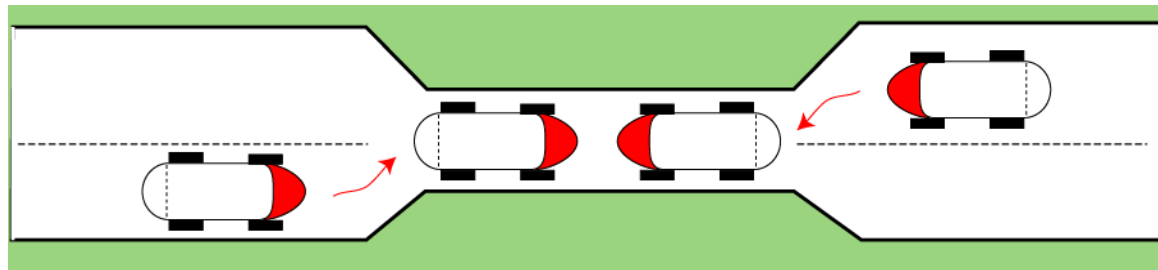
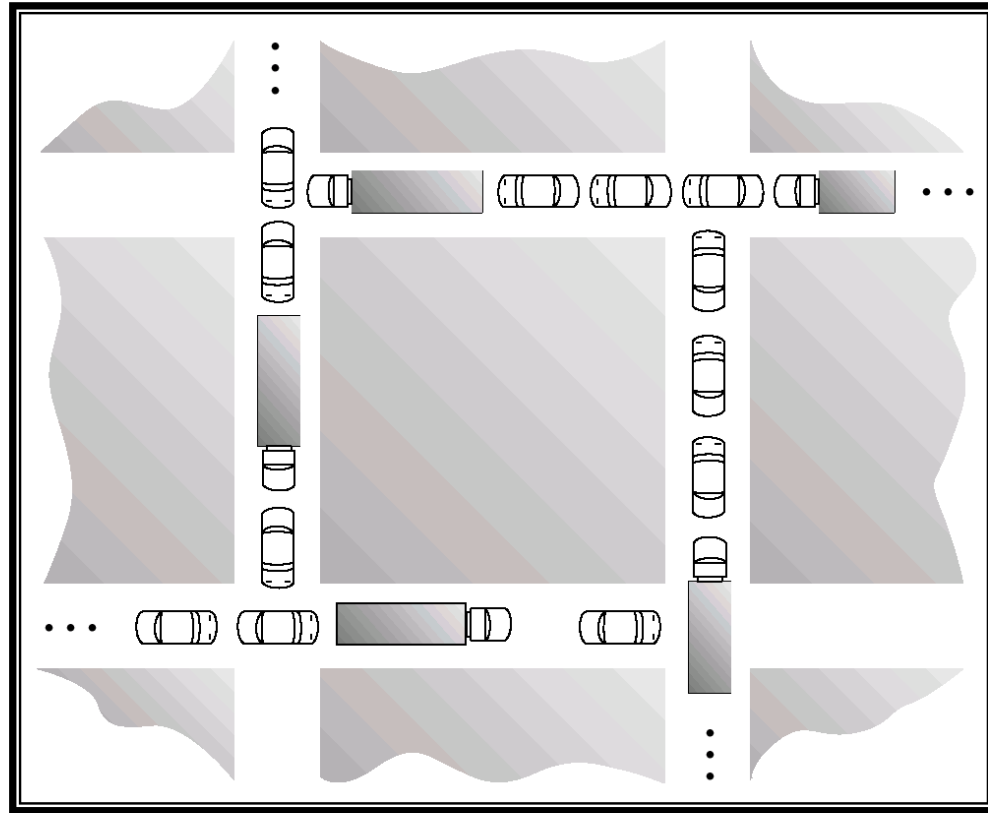
- ❖ 假設一個系統中有兩個資源 R_1 與 R_2 ，而且該系統產生出兩個行程 P_1 與 P_2

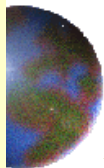


- ❖ R_1 已配置給 P_2 ， R_2 已配置給 P_1 ，但這兩個行程在執行過程中又向對方要求對方正在使用中的資源，因此造成系統打結。



生活上的死結範例





Dead Lock vs. Starvation

共同點：

- 皆為系統資源分配不當所造成

相異點：



Dead Lock	Starvation
由於一組 Processes 形成 Circular Waiting ，導致所有 Processes 無法往下執行。	由於單一 (或少數) Process 因長期取不到資源而形成 Infinite Blocking ，但其它 Processes 仍可正常運作。
CPU Utilization 及 Throughput 會大幅降低。	CPU Utilization 及 Throughput 不見得會大幅降低。
易發生在 Non-Preemptive 的環境	易發生在不公平、 Preemptive 的環境
解決：有三大處理方式	解決： Aging 技術



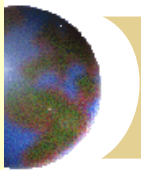


■ 形成Dead Lock的四個必要條件

⊕ 四個條件缺一不可：

- ⊗ **Mutual exclusion (互斥)**
- ⊗ **Hold and wait (持有並等待)**
- ⊗ **No preemption (不可搶先)**
- ⊗ **Circular waiting (循環式等候)**





❖ Mutual exclusion (互斥)

❖ **Def:** 某些資源在同一個時間點，最多只能被一個**Process**使用，不能有多個**Processes**同時使用此資源。其它欲使用此資源的**Process**則必須等待，直到該資源被釋放為止。

❖ e.g.: Printer, CPU

❖ 反例 : Read-only File

❖ Hold and wait (持有並等待)

❖ **Def:** **Process**持有部份資源，且在等待其它**Process**所持有的資源



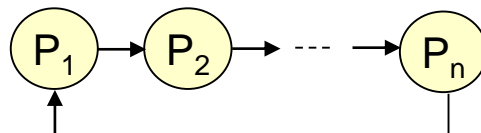


✚ No preemption (不可搶先)

- ✚ **Def:** **Process**不可任意搶奪其它**Process**所持有的資源。必須等待其它**Process**自願釋放這些資源才可以使用。

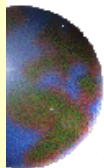
✚ Circular waiting (循環式等候)

- ✚ **Def:** 系統中存在一組**Processes** $\{P_0, P_1, \dots, P_n\}$, 其 P_0 正在等待 P_1 所持有的資源, P_1 正在等待 P_2 所持有的資源, ..., P_n 正在等待 P_0 所持有的資源, $P_0 \sim P_n$ 形成**Circular Waiting**。



- ✚ 因此, 死結不會發生在**Single process**環境中。





■ 死結的處理方法

⊕ **Dead Lock Prevention**

⊕ **Dead Lock Avoidance**

⊕ **Dead Lock Detection & Recovery**

優點：保證系統絕不會進入死結狀態。

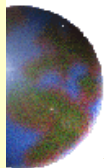
缺點：資源利用度低，系統**Throughput**低。

優點：資源利用度相對較高

缺點：①系統可能會發生死結，∴必須要偵測，若死結存在，則要做**Recovery**。

②**Cost**極高。





Dead Lock Prevention (死結預防)

✚ **觀念:** 打破四個必要條件其中之一，就可以保證死結永不發生。

✚ **作法:**

① 打破 “**Mutual Exclusion**”

- 很困難!!
- \therefore 互斥是某些資源與生俱來的性質, \therefore 無從下手打破!

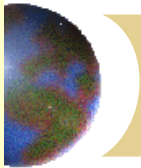
② 打破 “**No Preemptive**”

【作法】允許**Process**可以搶奪其它**Waiting Process**手中的資源。

(即: 改為**Preemptive**)

- 頂多只會有**Starvation**, 但**No Dead Lock**。(可以採用 “**Aging**” 技術解決)

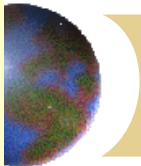




③ 打破 “Hold and Wait”

- 可採取下列作法之一：
 - a) 規定 “除非**Process**可以一次取得完成工作所需的全部資源，才允許**Process**持有資源，否則**Process**不准持有任何資源”。
 - 若一次取得全部資源 \Rightarrow “**Wait**”不成立；若無法取得全部資源，則全都不拿 \Rightarrow “**Hold**”不成立 (全有全無)
 - 系統產能低
 - b) **Process**在執行之初可持有部份資源，但若要再申請資源之前，必須先釋放手中所有資源，才可以提出申請。
 - 一開始可取部份資源執行 \Rightarrow “**Wait**”不成立；再申請時，需放掉手中所有資源 \Rightarrow “**Hold**”不成立
 - 系統產能低





④ 打破 “Circular Waiting”

- O.S.需採取下列措施：

a) 為每個不同類型的資源賦予一個獨一無二的資源編碼 (Unique ID)

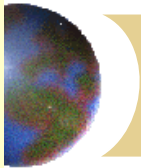
b) 規定Process必須按照資源編號遞增的方式提出資源申請

e.g.:

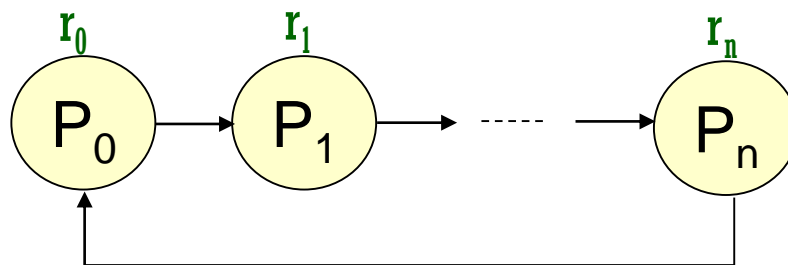
<u>Process持有資源</u>		<u>申請資源</u>	
R1, R2	→	R5	(✓)
R3	→	R1	(✗) 需先釋放R3方可申請
R1, R2, R6	→	R4	(✗) 需先釋放R6方可申請

如此系統不會有Circular Waiting存在



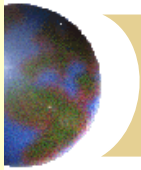


- ✚ 證明：假設在O.S.採取前述兩個措施後，系統仍存在一組Processes造成Circular Waiting如下：



- 令一組行程 $P_0 \sim P_n$ 分別持有資源 r_0, r_1, \dots, r_n ，且資源類型皆不相同(即：編號不同)。
- 根據遞增申請規則，上述Circular Waiting會推出 $r_0 < r_1 < \dots < r_n < r_0$ 。其中竟導出 $r_0 < r_0$ 此一矛盾的式子(即：相同的資源，其編號不同)。
- \therefore 假設不成立，不會有Circular Waiting狀況產生。

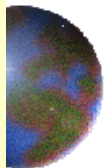




Summaries of the Dead Lock Prevention

- ✚ 優點: 保證系統絕不會有死結存在
- ✚ 缺點: **Resource Utilization**低, **Throughput**低





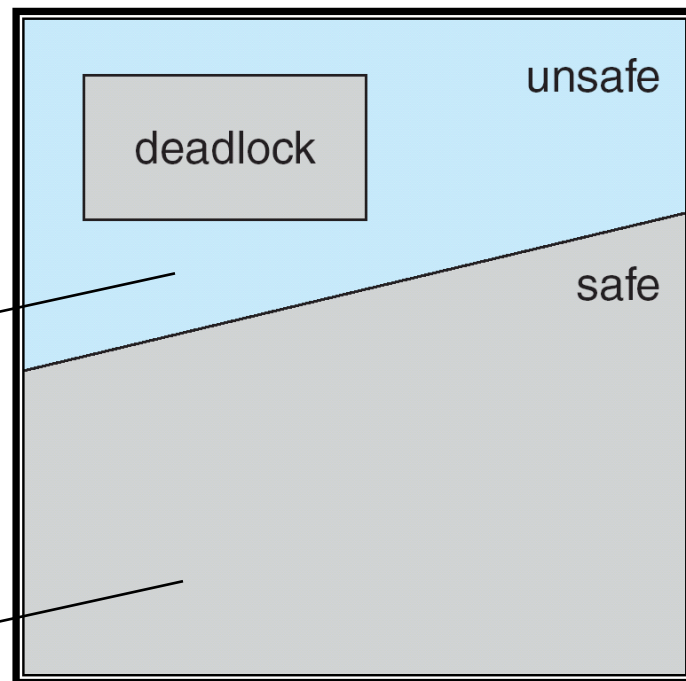
Dead Lock Avoidance (死結避免)

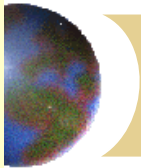
☉ **Def:** 當Process提出對資源的申請時, O.S.會根據以下資訊執行**銀行家演算法 (Banker's Algo., 內含Safety Algo.)**, 來判斷系統在“**假設**”核准申請後是否處於**Safe State**。若是, 則真的核准其請求; 否則否決此次申請, Process須再等待一段時間, 下一次再提出申請。

- ☒ 申請資源數量
- ☒ 各Process目前所持有的資源數量
- ☒ 各Process尚需要之資源需求量
- ☒ 系統目前可用的資源數量

不一定會有死結
(只要是Unsafe就不予配置,
縱使系統還不會Dead Lock)

一定不會有死結

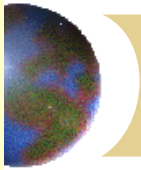




Banker's Algorithm (銀行家演算法)

- 針對提出資源申請的**Process**，來檢視系統“是否會因接受此一**Process**的申請而進入死結”。(內含**Safety Algorithm**)
- 所使用的資料結構：(假設系統目前有 n 個**Process**，與 m 種類型的資源)
 - Request _{i} [1... m]
 - 表示**Process** i 所提出的資源申請量
 - 若Request _{i} [j] = k ，則表示**Process** i 欲申請 k 個類型為 j 的資源
 - Allocation[1... n , 1... m]
 - 表示各**Process**目前持有的各類資源數量
 - 若Allocation[i , j] = k ，表示**Process** i 目前持有類型為 j 的資源 共 k 個
 - Max[1... n , 1... m]
 - 表示各**Process**需要哪些資源、且需要多少數量才得以完成工作 (即：記錄對各類資源的最大需求量)
 - 若Max[i , j] = k ，表示**Process** i 需要有類型為 j 的資源，且最多要 k 個方可完成工作





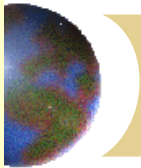
✦ **Need[1...n, 1...m]**

- 表示**Process** 目前尚需要多少數量的資源方得以完成工作
- 若**Need[i, j] = k**, 則表示**Process i** 尚需 類型為**j**的資源**k**個方能完成工作
- $\text{Need}_i = \text{Max}_i - \text{Allocation}_i$

✦ **Available[1...m]**

- 系統目前各類資源的可用數量
- **Available = 系統資源總量 - Allocation**

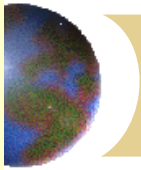




Banker's Algorithm之解題步驟

- ① 檢查 $\text{Request}_i \leq \text{Need}_i$
 - ✿ 即：檢查所提出的需求合不合理
 - ✿ 若不成立，則O.S.會視為illegal，中止此process；若成立，則Go to ②
- ② 檢查 $\text{Request}_i \leq \text{Available}$
 - ✿ 檢查系統是否有足夠資源可提供給Process
 - ✿ 若不成立，則process必須等待直到資源足夠；若成立，則Go to ③
- ③ (假設性試算) 假設系統分配資源給該提出申請之Process，透過計算下列數值以做接下來之安全演算法 (Safety Algorithm) 之分析。
 - ✿ $\text{Available} = \text{Available} - \text{Request}_i$ (分配後的系統可用資源還有多少)
 - ✿ $\text{Need}_i = \text{Need}_i - \text{Request}_i$ (分配後的Process i還需要多少資源才能完成工作)
 - ✿ $\text{Allocation}_i = \text{Allocation}_i + \text{Request}_i$ (分配後的Process i所掌握的資源有多少)
- ④ 執行Safety Algorithm。若系統判斷會處於Safe State，then允許申請；else否決此次申請，稍後再重新申請。





Safety Algorithm (安全演算法)

✪ 所使用的資料結構：(假設系統目前有 n 個 **Process**, 與 m 種類型的資源)

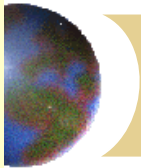
✪ **Work[1...m]**

- 當假定配置資源後, 目前系統可工作 (**Work**) 資源的數量累計
- 初值 = **Available**

✪ **Finish[1...n] of Boolean**

- **Finish[i]**表示**Process i**完成與否
 - **True**: 完成工作; **False**: 尚未完成
- 初值: **Finish[i] = False, $i = 1 \sim n$**
 - \therefore **Process**不可能一開始不取用任何資源就**Finish!!**





Safety Algorithm之解題步驟

① 設定初值

- ✚ **Work = Available** (分配後的系統可用資源還有多少。即：繼承前一個演算法的 **Available** 結果。)
- ✚ **Finish[i] = False, i = 1 to n**

② 找出一個 **Process i**, 滿足：

- ✚ **Need_i ≤ Work**
- ✚ **Finish[i] = False**

若找到, 則 **go to Step ③**; 否則 **go to Step ④**

③ 設定 **Finish[i] = True** 與 **Work = Work + Allocation i**, **go to Step ②**

④ 檢查 **Finish** 陣列, 若全部為 **True**, 則系統處於 **Safe State**, 否則處於 **Unsafe State**。

- ✚ 若可以找出至少一組 **Process** 執行順序, 讓所有 **Process** 完成, 此順序稱 **Safe Sequence**。(表示資源的分配、釋放 **OK**)



❖ 範例 ❖

🌀 [Note]: 通常在實際情況下, 下列資訊是已知的:

- ❖ 資源總量
- ❖ Max[]
- ❖ Allocation[]
- ❖ Request_i[]

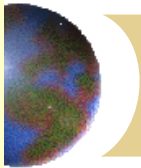
上述資訊可求出Need及Available。

🌀 假設系統內有5個Processes(P0~P4)及3種資源A, B, C, 其中A有10個, B有5個, C有7個。若系統目前狀態如下表所示。則:

❖ Need及Available的內容為何?

❖ 若P1提出Request₁[1, 0, 2], 則系統是否核准? (Using Banker's Algo.)

	Allocation			Max		
	A	B	C	A	B	C
P0	0	1	0	7	5	3
P1	2	0	0	3	2	2
P2	3	0	2	9	0	2
P3	2	1	1	2	2	2
P4	0	0	2	4	3	3



子問題1之解

- 找各**Process**尚需多少資源以完成工作。(即: **Need[]**)

❖ $\text{Need} = \text{Max} - \text{Allocation}$

❖

	Allocation			Max			Need		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3			
P1	2	0	0	3	2	2			
P2	3	0	2	9	0	2			
P3	2	1	1	2	2	2			
P4	0	0	2	4	3	3			

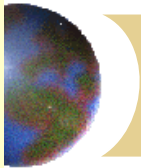
- 求系統目前各類資源的可用數量。(即: **Available[]**)

❖ 題目指出系統提供各類資源的總量分別為: **A = 10, B = 5, C = 7**

❖ 由上表可知, 目前各資源被**Process**所持有之總量分別為: **A = 0+2+3+2+0 = 7, B = 1+0+0+1+0 = 2, C = 0+0+2+1+2 = 5**

❖ 因此, 系統目前尚可提供的各類資源總量分別為: **A = 10-7 = 3; B = 5-2 = 3; C = 7-5 = 2**





✪ **P1**提出 $\text{Request}_1[1,0,2]$, 利用**Banker's Algo.**四個步驟來分析:

① 檢查 Request_i 是否小於等於 Need_i , 若成立則go to ②

- **P1**的 $\text{Request} = [1, 0, 2]$, $\text{Need} = [1, 2, 2]$, \therefore 成立

② 檢查 Request_i 是否小於等於 **Available**, 若成立則go to ③

- 系統目前的 $\text{Available} = [3, 3, 2]$, \therefore 成立

③ (假設性試算)。

- $\text{Available} = \text{Available} - \text{Request}_1 = [3, 3, 2] - [1, 0, 2] = [2, 3, 0]$
- $\text{Need}_1 = \text{Need}_1 - \text{Request}_1 = [1, 2, 2] - [1, 0, 2] = [0, 2, 0]$
- $\text{Allocation}_1 = \text{Allocation}_1 + \text{Request}_1 = [2, 0, 0] + [1, 0, 2] = [3, 0, 2]$

④ 執行**Safety Algo.**, 以判斷系統是否處於**Safe State**。若是**Safe**則核准; 否則不核准, 稍後再來申請。





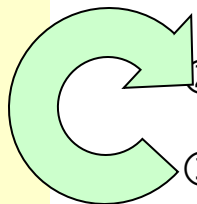
✿ 利用Safety Algorithm是否處於Safe State。

① 設定初值

- $Work = Available = [2, 3, 0]$

- 一維布林矩陣Finish[]:

0	1	2	3	4
F	F	F	F	F



② 找到P1, 滿足Finish[1] = False 且Need₁ ≤ Work, then go to ③

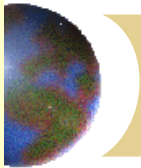
③ 設定Finish[1] = True, 且Work = Work + Allocation₁ = [5, 3, 2], then go to ②

[步驟②與③執行數次後, 可依序找到P3, P4, P0, P2皆滿足 (此序列不唯一)。且當執行完P2後再重執行步驟②時, 會因不滿足要件而go to ④]

④ 檢查Finish Array, 皆為True!! ∴系統處於Safe State → 核准P1之申請

✿ 上述推論找出一組Safe Sequence: **P1, P3, P4, P0, P2** (不只一組)





❖ 延伸範例 ❖

- ⊕ 接上題，若R4再提出[3, 3, 0]之請求，則是否核准？
 - ❑ 不核准
 - ❑ 當執行Banker's Algo.的步驟②時，會發現“檢查Request₄ ≤ Available”不成立!! 需令P4等待其它Process之資源釋放。(過程請自己算!!)
- ⊕ 若R0提出[0, 2, 0]之請求，則是否核准？
 - ❑ 不核准
 - ❑ 當執行到Safety Algo.時，會發現步驟④“檢查Finish陣列”並不皆為True，
∴系統處於Unsafe → 否決P0的申請。(過程請自己算!!)

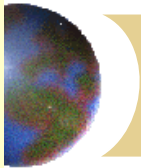




⊕ Banker's Algo.的優缺點:

- ❖ 優點: 避免系統發生死結的狀況
- ❖ 缺點: 此Algo.需要 $O(m \times n^2)$ 的時間複雜度 (m : 資源種類數; n : Process個數), 比較耗時。





Dead Lock Avoidance的重要定理

- ❖ 假設系統包含 m 個單一種類的**Resources**，且被 n 個**Process**共用。如果下列兩個條件滿足，則系統無死結存在 (**Dead Lock Free**)。
 - ① $1 \leq \text{Max}_i \leq m$
 - ② $\sum_{i=1}^n \text{Max}_i < m + n$
- ❖ 有6部印表機提供給 n 個process使用，每個process之最大需求量为2。在系統不發生**Dead Lock**的情況下，最多允許多少個process在系統內執行？(求 n 的最大值)

Ans:

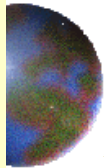
已知 $m = 6$, $\text{Max}_i = 2$

條件①滿足 ($\because 1 \leq \text{Max}_i = 2 \leq 6$)

欲滿足條件② (即: $\sum_{i=1}^n \text{Max}_i < m + n$), 則可得 $2n < 6 + n \rightarrow n < 6$

$\therefore n$ 的最大值為5。

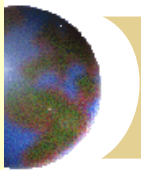




Dead Lock Detection & Recovery

- ✚ 若 **Dead Lock Prevention** 與 **Avoidance** 都不用，則系統中可能存在 **Dead Lock**。∴ 有必要提供下列機制：
 - ✚ **偵測** 死結是否存在
 - ✚ 若死結存在，則必須 **打破死結**，恢復正常
- ✚ 優點：
 - ✚ **Resource Utilization** 較高
 - ✚ **Throughput** 提升
- ✚ 缺點：
 - ✚ **Cost** 太高
- ✚ **Dead Lock Detection** 與 **Dead Lock Recovery** 是一體的





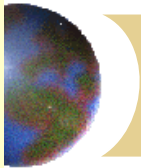
📍 Dead Lock Detection Algorithm.

❏ 偵測所有的Process, 看是否會進入死結 (勿和銀行家演算法搞混)

❏ Data Structures Used:

- Available[1...m]: 表示系統目前可用的資源數量
- Allocation[1...n, 1...m]: 表示各Process目前所持有的資源數量
- Request[1...n, 1...m]: 表示各個Process目前所提出的資源申請量
- Work[1...m]: 表示系統目前可用之資源數量之累計 (初值 = Available)
- Finish[1...n] of Boolean: 初始值設定規則為
 - If Allocation_i = 0, then Finish[i] = True (Process沒有持有任何資源, 即可假定它已完成)
 - If Allocation_i ≠ 0, then Finish[i] = False (尚未完成, 且Process持有資源)





處理步驟如下:

① 設定Work與Finish初始值

- **Work = Available**
- **Finish[i]** 的初值視**Process i** 是否持有資源而定
 - **True, if $\text{Allocation}_i = 0$**
 - **False, if $\text{Allocation}_i \neq 0$**

② 找到一個滿足下列兩條件的**Process P_i**。若找到, 則go to ③, 否則go to ④:

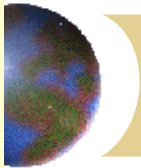
- **Finish[i] = False** (此**Process**尚未做完)
- **Request_i ≤ Work** (此**Process**當下所提出的申請, 系統可以應付)

③ 設定**Finish[i] = True**, **Work = Work + Allocation_i**, go to ②

④ 檢查**Finish**陣列

- 若皆為**True**, 則表示系統目前無死結。
- 若不是皆為**True**, 則表示有**Dead Lock**, 且**Finish[i] = False**者, 皆陷入此**Dead Lock**中。





❖ 範例 ❖

- ❖ 一個系統目前有五個處理程序 **P0~P4** 及三種資源 **A, B, C**。其中資源 **A** 有 **7** 個裝置，**B** 有 **2** 個裝置，**C** 有 **6** 個裝置，假設在時間 **T0** 時，系統的資源分配狀態如下所示：

	Allocation			Request			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	0	0	0	0	0	0
P1	2	0	0	2	0	2			
P2	3	0	3	0	0	0			
P3	2	1	1	1	0	0			
P4	0	0	2	0	0	2			

- ❖ 求系統目前有無死結？





Sol:

① 設定初值

✚ $Work = Available = (0, 0, 0)$

✚ $Finish =$

0	1	2	3	4
F	F	F	F	F

② 找到 P_i , 滿足 $Finish[i] = False$, 且 $Request_i \leq Work$

✚ 找到 P_0 , 滿足 $Finish[0] = False$, 且 $Request_0 \leq Work$ 。∴先找 P_0 , then go to ③。

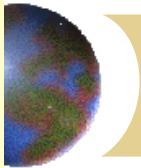
③ 設定 $Finish[i] = True$, 且 $Work = Work + Allocation_i$

✚ 設定 $Finish[0] = True$, 且 $Work = Work + Allocation_0 = (0, 0, 0) + (0, 1, 0) = (0, 1, 0)$ 。Go to ②

(步驟②與③交互執行, 可依序得到 P_2, P_1, P_3, P_4 滿足條件且完成工作)

④ 檢查 $Finish$ 陣列, 發現都為 $True$, ∴系統無Dead Lock





❖ 延伸範例 ❖

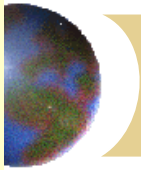
- ❖ 假設在時間T0時，系統的資源分配狀態如下所示：

	Allocation			Request			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	0	0	0	0	0	0
P1	2	0	0	2	0	2			
P2	3	0	3	0	0	1			
P3	2	1	1	1	0	0			
P4	0	0	2	0	0	2			

求系統目前有無死結？若有，哪些Process陷入死結？

- ❖ Ans: P0可完成工作，但P1, P2, P3, P4陷入死結





⊕ **Dead Lock Detection Algo. 的優點:**

- ⊕ 可以偵測出系統是否有死結存在。且若有死結, 可知哪些 **Processes** 陷入死結。

⊕ **Dead Lock Detection Algo. 的缺點:**

- ⊕ 此演算法須花費 $O(n^2m)$ 的時間複雜度 (**n: Process** 個數; **m: Resource** 種類)





Dead Lock Recovery

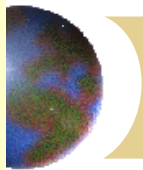
終止Process

- **Delete All**
 - ∴這些**Process**之前完成的工作全部白費! ∴成本太高
- 每次只終止一個**Process**, 直到**Dead Lock**打破為止
 - 每刪一個**Process**後皆需再執行**Dead Lock Detection Algo.**, 判斷有無死結
 - 若刪一個**Process**, **Dead Lock**仍在, 則表示該**Process**亦白殺
 - 成本亦高 (偵測、刪除都要成本)

資源搶奪

- 程序:
 - ① 選擇犧牲者**Process (Victim Process)**
 - ② 剝奪其資源
 - ③ 恢復到此**Victim process**原先無該資源的狀態 (**困難**)
 - **Cost**高!! ∴**OS**需要記錄每一個**Process**的每一次資源使用狀況
- 需考量**Starvation**問題之產生 (∴可搶奪)
 - **[解法]** 把被剝奪的次數列入選擇犧牲者**Process**之考量因素







Resource Allocation Graph (資源配置圖)

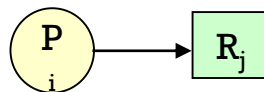
Def: 令 $G = (V, E)$ 為一有向圖，其中

V (頂點集合) 可分成兩類：

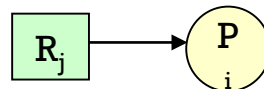
- **Process:** 通常以  表示。
- **Resource:** 通常以  表示，“•”表示該資源的數量。

E (邊集合) 也可分成兩類：

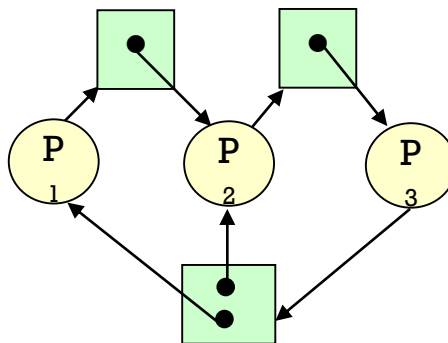
• **Request edge (申請邊)**



• **Allocated edge (配置邊)**



範例：

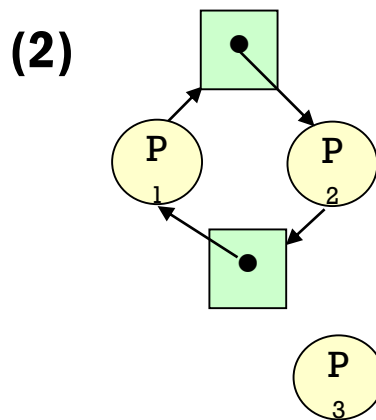
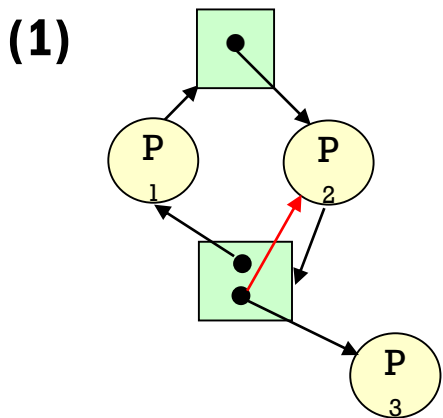


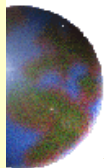


✧ 在Deadlock中重要觀念：

- ✧ 若圖形沒有**Cycle**存在，則系統無死結 (**No Cycle → No Deadlock**)
- ✧ 若系統中每類資源為**Multiple Instances** (多個數量)，則有**Cycle**存在，不一定有死結
- ✧ 若每類資源皆為**Single Instance**，則有**Cycle**存在，就一定有死結

✧ 範例：

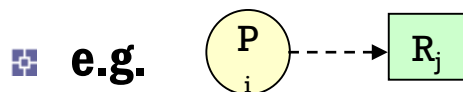




每個資源皆Single Instance的Deadlock Avoidance

作法：以Resource Allocation Graph為基礎

- ❖ 多加一種型態的edge, 叫做“**宣告邊 (Claim edge)**”, 表示 P_i 將來某時刻會對 R_j 提出申請 (但目前還未申請)。



- ❖ 在意義上, 有點類似銀行家演算法的Need資料結構, 表示“未來要完成此Process, 需要申請該Resource”。

當 P_i 真的提出對 R_j 的申請, 則執行下列步驟:

- ❖ 將宣告邊 改成申請邊

- ❖ 將申請邊 改成配置邊 (**假設性配置**)

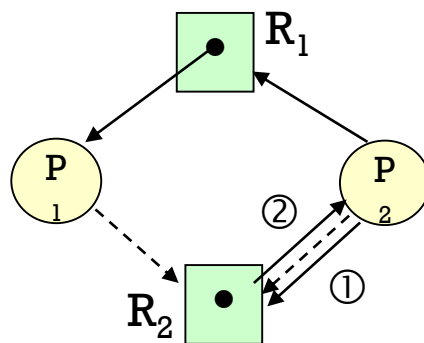
- ❖ 偵測此圖形是否有Cycle存在。若有, 則表示處於Unsafe狀況, 系統否決 P_i 的申請; 若沒有Cycle, 則表示處於Safe狀況, 核准 P_i 申請。

- 花費 $O(n^2)$, (假設使用相鄰矩陣來實做此圖形, n 為Process的數目)





❁ 範例：有一資源配置圖如下：

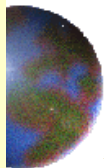


若 P_2 提出對 R_2 的申請，是否允許？

Ans:

檢查有**Cycle**存在， \therefore 否決此一申請。

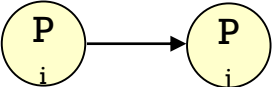





每個資源皆Single Instance的Deadlock Detection

✿ 作法：使用Wait-for Graph

✿ Def: 此圖是由Resource Allocation Graph演變而來。令 $G = \langle V, E \rangle$ 為一有向圖，其中：

- 頂點 V 是由Process組成。
- 邊 E 稱為Wait-for edge, 表示 “wait for” 的意思。
- Ex:  表示Process i 正在等待Process j 所持有的資源。

✿ 從Resource Allocation Graph轉換成Wait-for Graph的步驟：

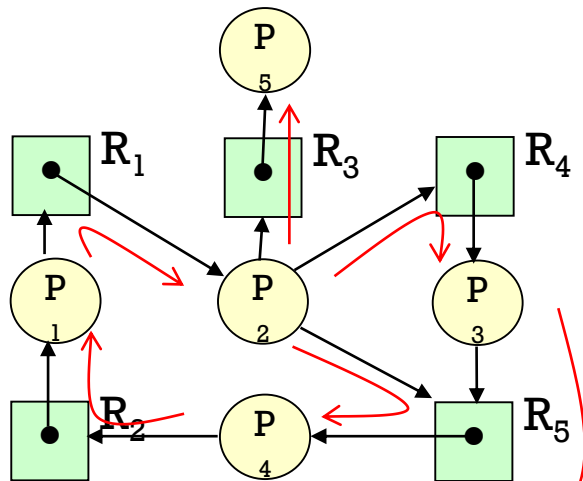
- ✿ 在Resource Allocation Graph中，找出 “ ”關係
- ✿ 將上述關係改成 “ ”



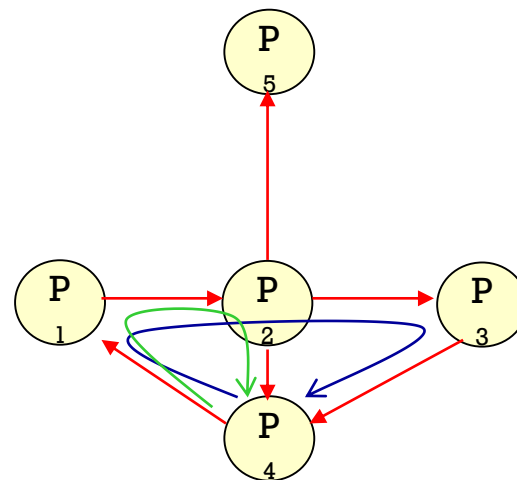
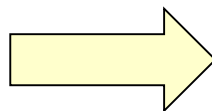


範例：

Resource Allocation Graph



Wait-for Graph



OS的偵測動作：

- 偵測Wait-for Graph是否有Cycle存在。若有，則表示有Dead Lock存在；否則不存在Dead Lock。

· ∵上例因存在兩個Cycle, ∴系統有Dead Lock

- 花費 $O(n^2)$